System $F_i$

a Higher-Order Polymorphic $\lambda$-Calculus with Erasable Term-Indices

Ki Yung Ahn$^1$, Tim Sheard$^1$, Marcelo Fiore$^2$, and Andrew M. Pitts$^2$

$^1$ Portland State University, Portland, Oregon, USA
$^2$ University of Cambridge, Cambridge, UK
$\{\text{kya,sheard}\}@\text{cs.pdx.edu}$
$\{\text{Marcelo.Fiore,Andrew.Pitts}\}@\text{cl.cam.ac.uk}$

Abstract. We introduce a foundational lambda calculus, System $F_i$, for studying programming languages with term-indexed datatypes – higher-kinded datatypes whose indices range over data such as natural numbers or lists. System $F_i$ is an extension of System $F_\omega$ that introduces the minimal features needed to support term-indexing. We show that System $F_i$ provides a theory for analysing programs with term-indexed types and also argue that it constitutes a basis for the design of logically-sound light-weight dependent programming languages. We establish erasure properties of $F_i$-types that capture the idea that term-indices are discardable in that they are irrelevant for computation. Index erasure projects typing in System $F_i$ to typing in System $F_\omega$. So, System $F_i$ inherits strong normalization and logical consistency from System $F_\omega$.

Keywords: term-indexed data types, generalized algebraic data types, higher-order polymorphism, type-constructor polymorphism, higher-kinded types, impredicative encoding, strong normalization, logical consistency

1 Introduction

We are interested in the use of indexed types to state and maintain program properties. A type parameter (like $\text{Int}$ in $(\text{List \ Int})$) usually tells us something about data stored in values of that type. A type-index (like $3$ in $(\text{Vector \ Int \ 3})$) states an inductive property of values with that type. For example, values of type $(\text{Vector \ Int \ 3})$ have three elements.

Indexed types come in two flavors: type-indexed and term-indexed types.

An example of type-indexing is a definition of a representation type [8] using GADTs in Haskell:

```
data TypeRep t where
  RepInt :: TypeRep Int
  RepBool :: TypeRep Bool
  RepPair :: TypeRep a -> TypeRep b -> TypeRep (a,b)
```

* supported by NSF grant 0910500.
Here, a value of type \(\text{TypeRep} \ t\) is isomorphic in shape with the type-index \(t\). For example, \((\text{RepPair} \ \text{RepInt} \ \text{RepBool}) :: \text{TypeRep} \ (\text{Int},\text{Bool})\).

An example of Term-indices are datatypes with indices ranging over data structures, such as natural numbers (like \(Z, (S \ Z)\)) or lists (like \(\text{Nil}\) or \((\text{Cons} \ Z \ \text{Nil})\)). A classic example of a term-index is the second parameter to the length-indexed list type \(\text{Vec}\) (as in \((\text{Vec} \ \text{Int} \ (S \ Z))\)).

In languages such as Haskell\(^3\) or OCaml [10], which support GADTs with only type-indexing, term-indices are simulated (or faked) by reflecting data at the type-level with uninhabited type constructors. For example,

\[
\begin{align*}
\text{data } S & \ n \\
\text{data } Z \\
\text{data } \text{Vec} & \ t \ n \ \text{where} \\
& \text{Cons} :: a \to \text{Vec} \ a \ n \to \text{Vec} \ a \ (S \ n) \\
& \text{Nil} :: \text{Vec} \ a \ Z
\end{align*}
\]

This simulation comes with a number of problems. First, there is no way to say that types such as \((S \ \text{Int})\) are ill-formed, and second the costs associated with duplicating the constructors of data to be used as term-indices. Nevertheless, GADTs with “faked” term-indices have become extremely popular as a lightweight, type-based mechanism to raise the confidence of users that software systems maintain important properties.

Our approach in this direction is to design a new foundational calculus, System \(F_i\), for functional programming languages with term-indexed datatypes. In a nutshell, System \(F_i\) is obtained by minimally extending System \(F_\omega\) with type-indexed kinds. Notably, this yields a logical calculus that is expressive enough to embed non-dependent term-indexed datatypes and their eliminators. Our contributions in this development are as follows.

- Identifying the features that are needed in a higher-order polymorphic \(\lambda\)-calculus to embed term-indexed datatypes (Sect. 2), in isolation from other features normally associated with such calculi (e.g., general recursion, large elimination, dependent types).
- The design of the calculus, System \(F_i\) (Sect. 4), and its use to study properties of languages with term-indexed datatypes, including the embedding of term-indexed datatypes into the calculus (Sect. 6) using Church or Mendler style encodings, and proofs about these encodings. For instance, one can use System \(F_i\) to prove that the Mendler-style eliminators for GADTs [3] are normalizing.
- Showing that System \(F_i\) enjoys a simple erasure property (Sect. 5.2) and inherits meta-theoretic results, strong normalization and logical consistency, from \(F_\omega\) (Sect. 5.3).

2 Motivation: from System \(F_\omega\) to System \(F_i\), and back

It is well known that datatypes can be embedded into polymorphic lambda calculi by means of functional encodings [5].

\(^3\) see Sect. 7 for a very recent GHC extension, which enable true term-indices.
In System F, one can embed regular datatypes, like homogeneous lists:

Haskell: \[ \text{data List } a = \text{Cons } a \text{ (List } a) \mid \text{Nil} \]
System F: \[ \text{List } A \triangleq \forall X. (A \to X \to X) \to X \to X \]
\[ \text{Cons } \triangleq \lambda w. \lambda x. \lambda y. \lambda z. y w (x y z), \text{Nil } \triangleq \lambda y. \lambda z. z \]

In such regular datatypes, constructors have algebraic structure that directly translates into polymorphic operations on abstract types as encapsulated by universal quantification over types (of kind \( \ast \)).

In the more expressive System \( F_\omega \) (where one can abstract over type constructors of any kind), one can encode more general type-indexed datatypes that go beyond the regular datatypes. For example, one can embed powerlists with heterogeneous elements in which an element of type \( a \) is followed by an element of the product type \( (a, a) \):

Haskell: \[ \text{data Powl } a = \text{PCons } a \text{ (Powl}(a, a)) \mid \text{PNil} \]
\[ \text{-- PCons } 1 \text{ (PCons } (2,3) \text{ (PCons } (3,4), (1,2) \text{ PNil)) } :: \text{Powl Int} \]
System \( F_\omega \): \[ \text{Powl } \triangleq \lambda A^\ast. \forall X^\ast. \forall Y^\ast. (A \to X (A \times A) \to X A) \to X A \to X A \]

Note the non-regular occurrence \( (\text{Powl}(a, a)) \) in the definition of \( (\text{Powl } a) \), and the use of universal quantification over higher-order kinds \( (\forall X^\ast Y^\ast) \). The term encodings for \( \text{PCons} \) and \( \text{PNil} \) are exactly the same as the term encodings for \( \text{Cons} \) and \( \text{Nil} \), but have different types.

What about term-indexed datatypes? What extensions to System \( F_\omega \) are needed to embed term-indices as well as type-indices? Our answer is System \( F_i \).

In a functional language supporting term-indexed datatypes, we envisage that the classic example of homogeneous length-indexed lists would be defined along the following lines (in Nax-like syntax):

Haskell: \[ \text{data Nat } = \text{S } \text{Nat} \mid \text{Z} \]
\[ \text{data Vec } : \ast \to \text{Nat} \to \ast \text{ where} \]
\[ \text{VCons } : a \to \text{Vec } a \{i\} \to \text{Vec } a \{S i\} \]
\[ \text{VNil } : \text{Vec } a \{Z\} \]

Here the type constructor \( \text{Vec} \) is defined to admit parameterisation by both type and term-indices. For instance, the type \( (\text{Vec } (\text{List } \text{Nat}) \{S (S Z)\}) \) is that of two-dimensional vectors of natural numbers. By design, our syntax directly reflects the difference between type and term-indexing by enclosing the latter in curly braces. We also make this distinction in System \( F_i \), where it is useful within the type system to guarantee the static nature of term-indexing.

The encoding of the vector datatype in System \( F_i \) is as follows:

\[ \text{Vec } \triangleq \lambda A^\ast. \lambda i^\ast. \forall X^\ast. (A \to X \{j\} \to X \{S j\}) \to X \{Z\} \to X \{i\} \]

where \( \text{Nat} \), \( \text{Z} \), and \( \text{S} \) respectively encode the natural number type and its two constructors, zero and successor. Again, the term encodings for \( \text{VCons} \) and \( \text{VNil} \) are exactly the same as the encodings for \( \text{Cons} \) and \( \text{Nil} \), but have different types.

Without going into the details of the formalism, which are given in the next section, one sees that such a calculus incorporating term-indexing structure needs four additional constructs (see Fig. 1 for the highlighted extended syntax).

\[ ^4 \text{We are developing a language called Nax whose theory is based on System } F_i. \]

---

4 We are developing a language called Nax whose theory is based on System \( F_i \).
1. Type-indexed kinding \( (A \rightarrow \kappa) \), as in \( (\text{Nat} \rightarrow *) \) in the example above, where the compile-time nature of term-indexing will be reflected in the typing rules, enforcing that \( A \) be a closed type (rule \((Ri)\) in Fig. 2).

2. Term-index abstraction \( \lambda i^A.F \) (as \( \lambda i^\text{Nat}. \cdot \cdot \cdot \) in the example above) for constructing (or introducing) term-indexed kinds (rule \((\lambda i)\) in Fig. 2).

3. Term-index application \( F\{s\} \) (as \( X\{2\}, X\{j\}, \text{and } X\{S j\} \) in the example above) for destructing (or eliminating) term-indexed kinds, where the compile-time nature of indexing will be reflected in the typing rules, enforcing that the index be statically typed (rule \((@i)\) in Fig. 2).

4. Term-index polymorphism \( \forall i^A.B \) (as \( \forall j^\text{Nat}. \cdot \cdot \cdot \) in the example above) where the compile-time nature of polymorphic term-indexing will be reflected in the typing rules enforcing that the variable \( i \) be static of closed type \( A \) (rule \((\forall Ii)\) in Fig. 2).

As described above, System \( F_i \) maintains a clear-cut separation between type-indexing and term-indexing. This adds a level of abstraction to System \( F_\omega \) and yields types that in addition to parametric polymorphism also keep track of inductive invariants using term-indices. All term-index information can be erased, since it is only used at compile-time. It is possible to project any well-typed System \( F_i \) term into a well-typed System \( F_\omega \) term. For instance, the erasure of the \( F_i \)-type \( \text{Vec} \) is the \( F_\omega \)-type \( \text{List} \). This is established in Sect. 5 and used to deduce the strong normalization of System \( F_i \).

3 Why Term-Indexed Calculi? (rather than dependent types)

We claim that a moderate extension to the polymorphic calculus \( (F_\omega) \) is a better candidate than a dependently typed calculus for the basis of a practical programming system. We hope to design a unified system for programming as well as reasoning. Language designs based on indexed types can benefit from existing compiler technology and type inference algorithms for functional programming languages. In addition, theories for term-indexed datatypes are simpler than theories for full-fledged dependent datatypes, because term-indexed datatypes can be encoded as functions (using Church-like encodings).

The implementation technology for functional programming languages based on polymorphic calculi is quite mature. The industrial strength Glasgow Haskell Compiler (GHC), whose intermediate core language is an extension of \( F_\omega \), is used by thousands every day. Our term-indexed calculus \( F_i \) is closely related to \( F_\omega \) by an index-erasure property. The hope is that a language implementation based on \( F_i \) can benefit from these technologies. We have built a language implementation of these ideas, which we call Nax.

Type inference algorithms for functional programming languages are often based on certain restrictions of the Curry-style polymorphic lambda calculi. These restrictions are designed to avoid higher-order unification during type inference. We have developed a conservative extension of Hindley–Milner type inference for Nax. This was possible because Nax is based on a restricted \( F_i \). Dependently typed languages, on the other hand, are often based on bidirectional
type checking, which requires annotations on top level definitions, rather than Hindley–Milner-style type inference.

In dependent type theories, datatypes are usually introduced as primitive constructs (with axioms), rather than as functional encodings (e.g., Church encodings). One can give functional encodings for datatypes in a dependent type theory, but one soon realizes that the induction principles (or, dependent eliminators) for those datatypes cannot be derived within the pure dependent calculi [11]. So, dependently typed reasoning systems support datatypes as primitives. For instance, Coq is based on Calculus of Inductive Constructions, which extends Calculus of Constructions [7] with dependent datatypes and their induction principles.

In contrast, in polymorphic type theories, all imaginable datatypes within the calculi have functional encodings (e.g., Church encodings). For instance, $F_\omega$ need not introduce datatypes as primitive constructs, since $F_\omega$ can embed all these datatypes, including non-regular recursive datatypes with type indices.

Another reason to use $F_i$ is to extend the application of Mendler-style recursion schemes, which are well-understood in the context of polymorphic lambda calculi like $F_\omega$. Researchers have thought about (though not published) a Mendler-style primitive recursion over dependently-typed functions over positive datatypes (i.e., datatypes that have a map), but not for negative (or, mixed-variant) datatypes. In System $F_i$, we can embed Mendler-style recursion schemes, (just as we embedded them in $F_\omega$) that are also well-defined for negative datatypes.

4 System $F_i$

System $F_i$ is a higher-order polymorphic lambda calculus designed to extend System $F_\omega$ by the inclusion of term-indices. The syntax and rules of System $F_i$ are described in Figs. 1, 2 and 3. The extensions new to System $F_i$, which are not originally part of System $F_\omega$, are highlighted by grey boxes. Eliding all the grey boxes from Figs. 1, 2 and 3, one obtains a version of System $F_\omega$ with Curry-style terms and the typing context separated into two parts (type-level context $\Delta$ and term-level context $\Gamma$).

We assume readers to be familiar with System $F_\omega$ and focus on describing the new constructs of $F_i$, which appear in grey boxes.

**Kinds** (Fig. 1). The key extension to $F_\omega$ is the addition of term-indexed arrow kinds of the form $A \rightarrow \kappa$. This allows type constructors to have terms as indices. The rest of the development of $F_i$ flows naturally from this single extension.

**Sorting** (Fig. 2). The formation of indexed arrow kinds is governed by the sorting rule $(\text{Ri})$. The rule $(\text{Ri})$ specifies that an indexed arrow kind $A \rightarrow \kappa$ is well-sorted when $A$ has kind $\ast$ under the empty type-level context $(\cdot)$ and $\kappa$ is well-sorted. Requiring $A$ to be well-kindled under the empty type-level context avoids

---

5 Tarmo Uustalu described this on a whiteboard when we met with him at the University of Cambridge in 2011.
**Syntax:**

Term Variables  \( x, y, z, \ldots, i, j, k, \ldots \)

Type Constructor Variables  \( X, Y, Z, \ldots \)

Sort  □

Kinds  \( \kappa ::= * \mid \kappa \to \kappa \mid A \to \kappa \)

Type Constructors  \( A, B, F, G ::= X \mid A \to B \mid \lambda X^\kappa.F \mid F G \mid \forall X^\kappa.B \)

Terms  \( r, s, t ::= x \mid \lambda x.t \mid r s \)

Typing Contexts  \( \Delta ::= \cdot \mid \Delta, X^\kappa \mid \Delta, i^A \)

\( \Gamma ::= \cdot \mid \Gamma, x : A \)

**Reduction:**

\[
\begin{align*}
(\lambda x.t) s & \rightsquigarrow t[s/x] & t \rightsquigarrow t' & \lambda x.t \rightsquigarrow \lambda x.t' & r \rightsquigarrow r' & r s \rightsquigarrow s s' & s \rightsquigarrow s'
\end{align*}
\]

**Fig. 1.** Syntax and Reduction rules of \( F_i \)

Dependent kinds (i.e., kinds depending on type-level or value-level bindings). That is, \( A \) should be a closed type of kind \( * \), which does not contain any free type variables or index variables. For example, \( (\text{List} X \to *) \) is not a well-sorted kind since \( X \) appears free, while \( ((\forall X^*. \text{List} X) \to *) \) is a well-sorted kind.

**Typing contexts** (Fig. 1). Typing contexts are split into two parts. Type level contexts \( (\Delta) \) for type-level (static) bindings, and term-level contexts \( (\Gamma) \) for term-level (dynamic) bindings. A new form of index variable binding \( (i^A) \) can appear in type-level contexts in addition to the traditional type variable bindings \( (X^\kappa) \). There is only one form of term-level binding \( (x : A) \) that appears in term-level contexts. Note, both \( x \) and \( i \) represent the same syntactic category of “Type Variables”. The distinction between \( x \) and \( i \) is only a convention for the sake of readability.

**Well-formed typing contexts** (Fig. 2). A type-level context \( \Delta \) is well-formed if (1) it is either empty, or (2) extended by a type variable binding \( X^\kappa \) whose kind \( \kappa \) is well-sorted under \( \Delta \), or (3) extended by an index binding \( i^A \) whose type \( A \) is well-kinded under the empty type-level context at kind \( * \). This restriction is similar to the one that occurs in the sorting rule \( (R_i) \) for term-indexed arrow kinds (see the paragraph *Sorting*). The consequence of this is that, in typing contexts and in sorts, \( A \) must be a closed type (not a type constructor!) without free variables.

A term-level context \( \Gamma \) is well-formed under a type-level context \( \Delta \) when it is either empty or extended by a term variable binding \( x : A \) whose type \( A \) is well-kinded under \( \Delta \).

**Type constructors and their kinding rules** (Figs. 1 and 2). We extend the type constructor syntax by three constructs, and extend the kinding rules accordingly.
Well-formed typing contexts:

\[
\begin{align*}
\Gamma & 
\vdash & \Delta & \vdash \kappa : \Box \\
\Delta & 
\vdash & \kappa : \Box & \quad (X \notin \text{dom}(\Delta)) \\
\Delta & 
\vdash & \kappa : \Box & \quad (i \notin \text{dom}(\Delta)) \\
\Delta & 
\vdash & \Lambda & \vdash A : * & \quad (x \notin \text{dom}(\Gamma))
\end{align*}
\]

Sorting:

\[
\begin{align*}
\vdash & \kappa : \\
\vdash & \kappa : \Box & \quad (A) \\
\vdash & \kappa : \Box & \quad (R) \\
\vdash & \kappa : \Box & \quad (Ri)
\end{align*}
\]

Kinding:

\[
\begin{align*}
\Delta & 
\vdash & F : \kappa & \quad (\text{Var}) \\
\Delta & 
\vdash & \lambda x^\kappa : F : \kappa' & \quad (\lambda) \\
\Delta & 
\vdash & \lambda x^\kappa : F : \kappa' & \quad (\lambda i) \\
\Delta & 
\vdash & \Delta, X^\kappa : \kappa' & \quad (\oplus) \\
\Delta & 
\vdash & \Delta, X^\kappa : \kappa' & \quad (\oplus i) \\
\Delta & 
\vdash & \Delta, X^\kappa : \kappa' & \quad (\forall) \\
\Delta & 
\vdash & \Delta, X^\kappa : \kappa' & \quad (\forall i)
\end{align*}
\]

Typing:

\[
\begin{align*}
\Delta, \Gamma & 
\vdash & t : A & \quad (i) \\
\Delta, \Gamma & 
\vdash & x : A & \quad (i) \\
\Delta, \Gamma & 
\vdash & \lambda x.t : A \rightarrow B & \quad (-\rightarrow I) \\
\Delta, \Gamma & 
\vdash & \lambda x.t : A \rightarrow B & \quad (-\rightarrow E) \\
\Delta, \Gamma & 
\vdash & t : A \rightarrow B & \quad (\forall I) \\
\Delta, \Gamma & 
\vdash & \forall X^\kappa.B & \quad (\forall E) \\
\Delta, \Gamma & 
\vdash & t : \forall X^\kappa.B & \quad (\forall i I) \\
\Delta, \Gamma & 
\vdash & B[\{s/i\}] & \quad (\forall E i)
\end{align*}
\]

Fig. 2. Well-formedness, Sorting, Kinding, and Typing rules of \( \mathcal{F}_i \)
Kind equality: 
\[ \vdash \kappa = \kappa' : \Box \]
\[ \vdash A = A' : * \vdash \kappa = \kappa' : \Box \]
\[ \vdash A \to \kappa = A' \to \kappa' : \Box \]

Type constructor equality: 
\[ \Delta \vdash F = F' : \kappa \]
\[ \Delta, X^\kappa \vdash F : \kappa' \Delta \vdash G : \kappa \]
\[ \Delta \vdash (\lambda X^\kappa . F) G = F[G/X] : \kappa' \]
\[ \Delta, i^A \vdash F : \kappa \Delta \vdash G : \kappa \]
\[ \Delta \vdash (\lambda i^A . F) \{ s \} = F \{ s[i] \} : \kappa \]
\[ \Delta \vdash F = F' : A \to \kappa \Delta ; \vdash s = s' : A \]
\[ \Delta \vdash F \{ s \} = F' \{ s' \} : \kappa \]

Term equality: 
\[ \Delta ; \Gamma \vdash t = t' : A \]
\[ \Delta ; \Gamma, x : A \vdash t : B \Delta ; \Gamma \vdash s : A \]
\[ \Delta ; \Gamma \vdash (\lambda x . t) s = t[s/x] : B \]

Fig. 3. Equality rules of $F_i$ (only the key rules are shown)

$\lambda i^A F$ is the type-level abstraction over an index (or, index abstraction). Index abstractions introduce indexed arrow kinds by the kinding rule $(\lambda i)$. Note, the use of the new form of context extension, $i^A$, in the kinding rule $(\lambda i)$.

$F \{ s \}$ is the type-level term-index application. In contrast to the ordinary type-level type-application ($FG$) where the argument ($G$) is a type (of arbitrary kind). The argument of an term-index application ($F \{ s \}$) is a term ($s$). We use the curly bracket notation around an index argument in a type to emphasize the transition from ordinary type to term, and to emphasize that $s$ is a term-index, which is erasable. Index applications eliminate indexed arrow kinds by the kinding rule $(\lambda i)$. Note, we type check the term-index ($s$) under the current type-level context paired with the empty term-level context ($\Delta ; \cdot$) since we do not want the term-index ($s$) to depend on any term-level bindings. Otherwise, we would admit value dependencies in types.

$\forall i^A B$ is an index polymorphic type. The formation of indexed polymorphic types is governed by the kinding rule $(\forall i)$, which is very similar to the formation rule $(\forall)$ for ordinary polymorphic types.

In addition to the rules $(\lambda i)$, $(\lambda i)$, and $(\forall i)$, we need a conversion rule $(\text{Conv})$ at kind level. This is because the new extension to the kind syntax $A \to \kappa$ involves types. Since kind syntax involves types, we need more than simple structural equality over kinds (see Fig. 3). For instance, $A \to \kappa$ and $A' \to \kappa$ equivalent kinds when $A$ and $A'$ are equivalent types. Only the key equality rules are shown in Fig. 3, and the other structural rules (one for each sorting/kind/typing rule) and the congruence rules (symmetry, transitivity) are omitted.

Terms and their typing rules (Figs. 1 and 2). The term syntax is exactly the same as other Curry-style calculi. We write $x$ for ordinary term variables introduced by term-level abstractions $(\lambda x . t)$. We write $i$ for index variables introduced...
by index abstractions ($\lambda i^A.F$) and by index polymorphic types ($\forall i^A.B$). As discussed earlier, the distinction between $x$ and $i$ is only for readability.

Since $F_i$ has index polymorphic types ($\forall i^A.B$), we need typing rules for index polymorphism: ($\forall Ii$) for index generalization and ($\forall Ei$) for index instantiation. These rules are similar to the type generalization ($\forall I$) and the type instantiation ($\forall I$) rules, but involve indices, rather than types, and have additional side conditions compared to their type counterparts.

The additional side condition $i \notin \text{FV}(t)$ in the ($\forall Ii$) rule prevents terms from accessing the type-level index variables introduced by index polymorphism. Without this side condition, $\forall$-binder would no longer behave polymorphically, but instead would behave as a dependent function binder, which are usually noted by $\Pi$ in dependent type theories. Such side conditions on generalization rules for polymorphism are fairly standard in dependent type theories that distinguish between polymorphism (or, erasable arguments) and dependent functions (e.g., IPTS[17], ICC[16]).

The index instantiation rule ($\forall Ei$) requires that the term-index $s$, which instantiates $i$, be well-typed in the current type-level context paired with the empty term-level context ($\Delta; \cdot$) rather than the current term-level context, since we do not want indices to depend on term-level bindings.

In addition to the rules ($\forall Ii$) and ($\forall Ei$) for index polymorphism, we need an additional variable rule ($\cdot ; i$) to access index variables already in scope. In examples like ($\lambda i^A.F\{s\}$) and ($\forall i^A.F\{s\}$), the term ($s$) should be able to access the index variable ($i$) already in scope.

5 Metatheory

The expectation is that System $F_i$ has all the nice properties of System $F_\omega$, yet is more expressive (i.e., can state finer grained program properties) because of the addition of term-indexed types.

We show some basic well-formedness properties for the judgments of $F_i$ in Sect. 5.1. We prove erasure properties of $F_i$, which capture the idea that indices are erasable since they are irrelevant for reduction in Sect. 5.2. We show strong normalization, logical consistence, and subject reduction for $F_i$ by reasoning about well-known calculi related to $F_i$ in Sect. 5.3.

5.1 Well-formedness and Substitution Lemmas

We want to show that kinding and typing derivations give well-formed results under well-formed contexts. That is, kinding derivations ($\Delta \vdash F : \kappa$) result in well-sorted kinds ($\vdash \kappa$) under well-formed type-level contexts ($\vdash \Delta$) (Proposition 1), and typing derivations ($\Delta; \Gamma \vdash t : A$) result in well-kindred types ($\Delta; \Gamma \vdash A : *$) under well-formed type and term-level contexts (Proposition 2).

Proposition 1. \[ \text{if } \vdash \Delta \text{ then } \Delta \vdash F : \kappa \]

Proposition 2. \[ \Delta \vdash \Gamma \text{ then } \Delta; \Gamma \vdash t : A \]
We can prove these well-formedness properties by induction over the judgment and using the substitution lemma below.

**Lemma 1 (substitution).**

1. **(type substitution)** \[ \Delta, X \vdash F : \kappa \quad \Delta \vdash G : \kappa \]
   \[ \Delta \vdash F[G/X] : \kappa' \]

2. **(index substitution)** \[ \Delta, i^A \vdash F : \kappa \quad \Delta; i \vdash s : A \]
   \[ \Delta \vdash F[s/i] : \kappa \]

3. **(term substitution)** \[ \Delta; \Gamma, x : A \vdash t : B \quad \Delta; \Gamma \vdash s : A \]
   \[ \Delta; \Gamma \vdash t[s/x] : B \]

These substitution lemmas are fairly standard, comparable to substitution lemmas in other well-known systems such as $\text{F}_\omega$ or ICC.

### 5.2 Erasure Properties

We define a meta-operation of index erasure that projects $F_i$-types to $F_\omega$-types.

**Definition 1 (index erasure).**

\[
\begin{align*}
\kappa^o &= * \\
F^o &= X \\
(\lambda X^\kappa.F)^o &= \lambda X^{\kappa^o}.F^o \\
(F G)^o &= F^o G^o \\
(\forall X^\kappa . B)^o &= \forall X^{\kappa^o} . B^o \\
\Delta^o &= \cdot \\
(\Delta, X^\kappa)^o &= \Delta^o, X^{\kappa^o} \\
(\Delta, i^A)^o &= \Delta^o, i : A \\
\Gamma^o &= \cdot \\
(\Gamma, x : A)^o &= \Gamma^o, x : A^o 
\end{align*}
\]

In addition, we define another meta-operation, which selects out all the index variable bindings from the type-level context. We use this in Theorem 6.

**Definition 2 (index variable selection).**

\[
\begin{align*}
\Delta^* &= \cdot \\
(\Delta, X^\kappa)^* &= \Delta^*, X^{\kappa^*} \\
(\Delta, i^A)^* &= \Delta^*, i : A 
\end{align*}
\]

**Theorem 1 (index erasure on well-sorted kinds).**

\[ \vdash \kappa : \Box \]

\[ \vdash \kappa^o : \Box \]

**Proof.** By induction on the sort ($\kappa$). \[ \square \]

---

6 The proof for Propositions 1 and 2 are mutually inductive. So, we prove these two propositions at the same time, using a combined judgment $J$, which is either a kinding judgment or a typing judgment (i.e., $J := \Delta \vdash F : \kappa | \Delta; \Gamma \vdash t : A$).
Remark 1. For any well-sorted kind $\kappa$ in $F_i$, $\kappa^\circ$ is a well-sorted kind in $F_\omega$.

Theorem 2 (index erasure on well-formed type-level contexts). \[ \Delta \vdash F : \kappa \] \[ \Delta^\circ \vdash F^\circ : \kappa^\circ \]

Proof. By induction on the type-level context $\Delta$ and using Theorem 1. 

Remark 2. For any well-formed type-level context $\Delta$ in $F_i$, $\Delta^\circ$ is a well-formed type-level context in $F_\omega$.

Theorem 3 (index erasure on kind equality). \[ \vdash \kappa = \kappa' : \Box \] \[ \vdash \kappa^\circ = \kappa'^\circ : \Box \]

Proof. By induction on the kind equality derivation $\vdash \kappa = \kappa' : \Box$. 

Remark 3. For any well-sorted kind equality $\vdash \kappa = \kappa' : \Box$ in $F_i$, $\kappa^\circ$ and $\kappa'^\circ$ are the syntactically same $F_\omega$ kinds. Note that no variables can appear in the erased kinds by definition of the erasure operation on kinds.

Theorem 4 (index erasure on well-kinded type constructors). \[ \Delta \vdash F : \kappa \] \[ \Delta^\circ \vdash F^\circ : \kappa^\circ \]

Proof. By induction on the kinding derivation $\Delta \vdash F : \kappa$. We use Theorem 2 in the (Var) case, Theorem 3 in the (Conv) case, and Theorem 1 in the ($\lambda$) and ($\forall$) cases. 

Remark 4. In the theorem above, $F^\circ$ is a well-kinded type constructor in $F_\omega$.

Lemma 2. $(F[G/X])^\circ = F^\circ[G^\circ/X]$ 
Lemma 3. $(F[s/i])^\circ = F^\circ$

Theorem 5 (index erasure on type constructor equality). \[ \Delta \vdash F = F' : \kappa \] \[ \Delta^\circ \vdash F^\circ = F'^\circ : \kappa^\circ \]

Proof. By induction on the derivation of the type constructor equality judgment $\Delta \vdash F = F' : \kappa$. We also use Proposition 1 and Lemmas 2 and 3. 

Remark 5. When $\Delta \vdash F = F' : \kappa$ is a valid type constructor equality in $F_i$, $\Delta^\circ \vdash F^\circ = F'^\circ : \kappa^\circ$ is a valid type constructor equality in $F_\omega$.

Theorem 6 (index erasure on well-formed term-level contexts prepended by index variable selection). \[ \Delta \vdash \Gamma \] \[ \Delta^\circ \vdash (\Gamma^\bullet, \Gamma)^\circ \]

Proof. By induction on the term-level context $\Gamma$ and using Theorem 4. 

Remark 6. We can also show that $\Delta \vdash \Gamma \vdash \Gamma^\circ$ and prove Corollary 1 directly.
Theorem 7 (index erasure on well-typed terms).

\[ \Delta \vdash \Gamma \quad \Delta ; \Gamma \vdash t : A \]

\[ \Delta^0 ; (\Delta^*, \Gamma^0) \vdash t : A^0 \]

**Proof.** By induction on the typing derivation \((\Delta; \Gamma \vdash t : A)\). We also make use of Theorems 1, 4, 5, and 6.

**Remark 7.** In the theorem above, \(t\) is a well typed term in \(F_\omega\) as well as in \(F_i\).

Corollary 1 (index erasure on index-free well-typed terms).

\[ \Delta \vdash \Gamma \quad \Delta ; \Gamma \vdash t : A \]

\[ \Delta^0 ; \Gamma^0 \vdash t : A^0 \quad (\text{dom}(\Delta) \cap \text{FV}(t) = \emptyset) \]

### 5.3 Strong Normalization and Logical Consistency

Strong normalization is a corollary of the erasure property since we know that System \(F_\omega\) is strongly normalizing. Index erasure also implies logical consistency. By index erasure, we know that any well-typed term in \(F_i\) is a well-typed term in \(F_\omega\) with its erased type. That is, there are no extra well-typed terms in \(F_i\) that are not well-typed in \(F_\omega\). By the saturated sets model (as in [1]), we know that the void type \((\forall X^*.X)\) in \(F_\omega\) is uninhabited. Therefore, the void type \((\forall X^*.X)\) in \(F_i\) is uninhabited since it erases to the same void type in \(F_\omega\). Alternatively, logical consistency of \(F_i\) can be drawn from ICC. System \(F_i\) is a restriction of the **restricted implicit calculus** [15] or ICC\(^-\) [4], which are restrictions of ICC [16] known to be logically consistent.

### 6 Encodings of Term-Indexed Datatypes

Recall that our motivation was a foundational calculus that can encode term-indexed datatypes. In Sect. 2, we gave Church encodings of \textit{List} (a regular datatype), \textit{Powl} (a type-indexed datatype), and \textit{Vec} (a term-indexed datatype). In this section, we discuss a more complex datatype [6] involving nested term-indices, and several encoding schemes that we have seen used in practice – first, encoding indexed datatypes using equality constraints [18, 8] and second, encoding datatypes in the Mendler-style [2, 3].

**Nested term-indices:** System \(F_i\) is able to express datatypes with *nested term-indices* – term-indices which are themselves term-indexed datatypes. Consider the resource-state tracking environment [6] in Nax-like syntax below:

```nax
data Env : ({st} -> *) -> {Vec st {n}} -> * where
  Extend : res {x} -> Env res {xs} -> Env res {VCons x xs}
  Empty : Env res {VNil}
```

Note that \textit{Env} has a term-index of type \textit{Vec}, which is again indexed by \textit{Nat}. For simplicity,\(^7\) assume that \(n\) is some fixed constant (e.g., \(S(S(Z))\), i.e., 3). Then,

\(\text{Nax}\) supports rank-1 kind-level polymorphism. It would be virtually useless if nested term-indices were only limited to constants rather than polymorphic variables. We

\(^7\) Nax supports rank-1 kind-level polymorphism. It would be virtually useless if nested term-indices were only limited to constants rather than polymorphic variables. We
Encoding indexed datatypes using equality constraints: Systematic encodings of GADTs [18, 8], which are used in practical implementations, typically involve equality constraints and existential quantification. Here, we want to emphasize that such encoding schemes are expressible within System F, since it is possible to define equalities and existentials over both types and term-indices in F.

It is well known that Leibniz equality over type constructors can be defined within System Fω as $(\equiv) \triangleq \lambda X^\kappa. \lambda X_1^\kappa. \forall X^{\kappa \rightarrow \tau}. X X_1 \rightarrow X X_2$. Similarly, Leibniz equality over term-indices is defined as $(\equiv^) \triangleq \lambda i^A. \lambda j^A. \forall X^{A \rightarrow \tau}. X \{i\} \rightarrow X \{j\}$ in System F. Then, we can encode Vec as the sum of its two data constructor types:

$$
\text{Vec} \triangleq \lambda A^\kappa. \lambda i^{\text{Nat}}. \forall X^{\kappa \rightarrow \tau}. (\exists j^{\text{Nat}}. (S j \equiv i) \times A \times X \{j\}) + (Z \equiv i)
$$

where + and × are the usual impredicative encoding of sums and products. We can encode the existential quantification over indices (used in the encoding of Vec above) as $\exists^A. B \equiv \forall X^{\tau}. (\forall i^A. B \rightarrow X) \rightarrow X$, which is similar to the usual encoding of the existential quantification over types in System F or Fω.

Compared to the simple Church encoded versions in Sect. 2, the encodings using equality constraints work particularly well with encodings of functions that constrain their domain types by restricting their indices. For instance, the function safeTail : Vec a {S n} → Vec a {n}, which can only be applied to non-empty length indexed lists due the index of the domain type (S n).

The Mendler-style encoding: Recursive type theories that extend higher-order polymorphic lambda calculi typically come with a built-in recursive type operator $\mu_i : (\kappa \rightarrow \kappa) \rightarrow \kappa$ for each kind $\kappa$, which yields recursive types $(\mu_i F : \kappa)$ when applied to type constructors of appropriate kind $(F : \kappa \rightarrow \kappa)$. For instance, List $\triangleq \lambda Y^\kappa. \mu_s (\lambda X^\kappa. Y \times X + 1)$ where I is the unit type. One benefit of factoring out the recursion at type-level (e.g., $\mu_s$) from the base structure (e.g., $\lambda X^\kappa. Y \times X + 1$) of recursive types is that such factorized (or, two-level) representations are more amenable to express generic recursion schemes (e.g., catamorphism) that work over different recursive datatypes. Interestingly, there exists an encoding scheme, namely the Mendler style, which can embed $\mu_s$ within Systems like Fω or F. The Mendler-style encoding can keep the theoretical basis small, while enjoying the benefits of factoring out the recursion at type-level.

Strangely believe rank-1 kind polymorphism does not introduce inconsistency, since rank-1 polymorphic systems are essentially equivalent to simply-typed systems by inlining the polymorphic definition with the instantiated arguments in each instantiation site.
7 Related Work

System \( F_i \) is most closely related to Curry-style System \( F_\omega \) \[2, 12\] and the Implicit Calculus of Constructions (ICC) \[16\]. All terms typable in a Curry-style System \( F_\omega \) are typable (with the same type) in System \( F_i \) and all terms typable in \( F_i \) are typable (with the same type\(^8\)) in ICC.

As mentioned in Sect. 5.3, we can derive strong normalization of \( F_i \) from System \( F_\omega \), and derive logical consistency of \( F_i \) from certain restrictions of ICC \[15, 4\]. In fact, ICC is more than just an extension of System \( F_i \) with dependent types and stratified universes, since ICC includes \( \eta \)-reduction and \( \eta \)-equivalence. We do not foresee any problems adding \( \eta \)-reduction and \( \eta \)-equivalence to System \( F_i \). Although System \( F_i \) accepts fewer terms than ICC, it enjoys simpler erasure properties (Theorem 7 and Corollary 1) just by looking at the syntax of kinds and types, which ICC cannot enjoy due to its support for full dependent types. In System \( F_i \), term-indices appearing in types (e.g., \( s \) in \( F\{s\} \)) are always erasable. Mishra-Linger and Sheard \[17\] generalized the ICC framework to one which describes erasure on arbitrary Church-style calculi (EPTS) and Curry-style calculi (IPTS), but only consider \( \beta \)-equivalence for type conversion.

In the practical setting of programming language implementation, Yorgey et al. \[19\], inspired by McBride \[14\], recently designed an extension to Haskell’s GADTs by allowing datatypes to be used as kinds. For instance, \( \text{Bool} \) is promoted to a kind (i.e., \( \text{Bool} : \Box \)) and its data constructors \( \text{True} \) and \( \text{False} \) are promoted to types. They extended System \( F_C \) (the Glasgow Haskell Compiler’s intermediate core language) to support \textit{datatype promotion} and named it System \( F_i^C \). The key difference between \( F_i^C \) and \( F_i \) is in their kind syntax:

\[
F_i^C \textit{kinds } \kappa ::= \ast | \kappa \to \kappa | F\kappa | \forall X.\kappa | \cdots \\
F_i \textit{kinds } \kappa ::= \ast | \kappa \to \kappa | A \to \kappa
\]

In \( F_i^C \), all type constructors \((F)\) are promotable to the kind level and become kinds when fully applied to other kinds \((F\kappa)\). On the other hand, in \( F_i \), a type can only appear as the domain of an index arrow kind \((A \to \kappa)\). The ramifications of this difference is that \( F_i^C \) can express type-level data structures but not nested term-indices, while \( F_i \) supports the converse. Intuitively, a type constructor like \( \text{List} : \ast \to \ast \) is promoted to a kind constructor \( \text{List} : \Box \to \Box \), which enables type-level data structures such as \([\text{Nat}, \text{Bool}, \text{Nat} \to \text{Bool}] : \text{List}\ast\). Type-level data structures motivate type-level computations over promoted data. This is made possible by type families\(^9\). The promotion of polymorphic types naturally motivates kind polymorphism \((\forall X.\kappa)\). Kind polymorphism of arbitrary rank is known to break strong normalization and cause logical inconsistency \[13\]. In a \textit{programming language}, inconsistency is not an issue. However, when studying logically consistent systems, we need a more conservative approach, as in \( F_i \).

\(^8\) The \( \ast \) kind in \( F_\omega \) and \( F_i \) corresponds to \( \text{Set} \) in ICC

\(^9\) A GHC extension to define type-level functions in Haskell.
System $F_i$ is a strongly-normalizing, logically-consistent, higher-order polymorphic lambda calculus that was designed to support the definition of datatypes indexed by both terms and types. In terms of expressivity, System $F_i$ sits between System $F_{\omega}$ and ICC. We designed System $F_i$ as a tool to reason about programming languages with term-indexed datatypes. System $F_i$ can express a large class of term-indexed datatypes, including datatypes with nested term-indices.

One limitation of System $F_i$ is that it cannot express type-level data structures such as lists that contain type elements. We hope to overcome this limitation by extending $F_i$ with first-class type representations [9], which reflect types as term-level data (a sort of a fully reflective version of TypeRep from Sect. 1).

Our goal is to build a unified programming and reasoning system, which supports (1) an expressive class of datatypes including nested term-indexed datatypes and negative datatypes, (2) logically consistent reasoning about program properties, and (3) Hindley–Milner-style type inference. Towards this goal, we are developing the programming language Nax based on System $F_i$. Nax is given semantics in terms of System $F_i$. That is, all the primitive language constructs of Nax that are not present in $F_i$ have translations into System $F_i$. Such constructs include Mendler-style eliminators, recursive type operators, and pattern matching.

Some language features we want to include in Nax go beyond $F_i$. One of them is a recursion scheme that guarantee normalization due to paradigmatic use of indices in datatypes. For instance, some recursive computations always reduce a natural number term-index in every recursive call. Although such computations obviously terminate, we cannot express them in System $F_i$, since term-indices in them are erasable – $F_i$ only accepts terms that are already type-correct in $F_{\omega}$. We plan to explore extensions to System $F_i$ that enable such computations while maintaining logical consistency.
Bibliography